Writing Trusted Applications

Ari Singer

NTRU Cryptosystems

September 12, 2005

EmbeddedSystems Conferences San Francisco Boston Asia Munich

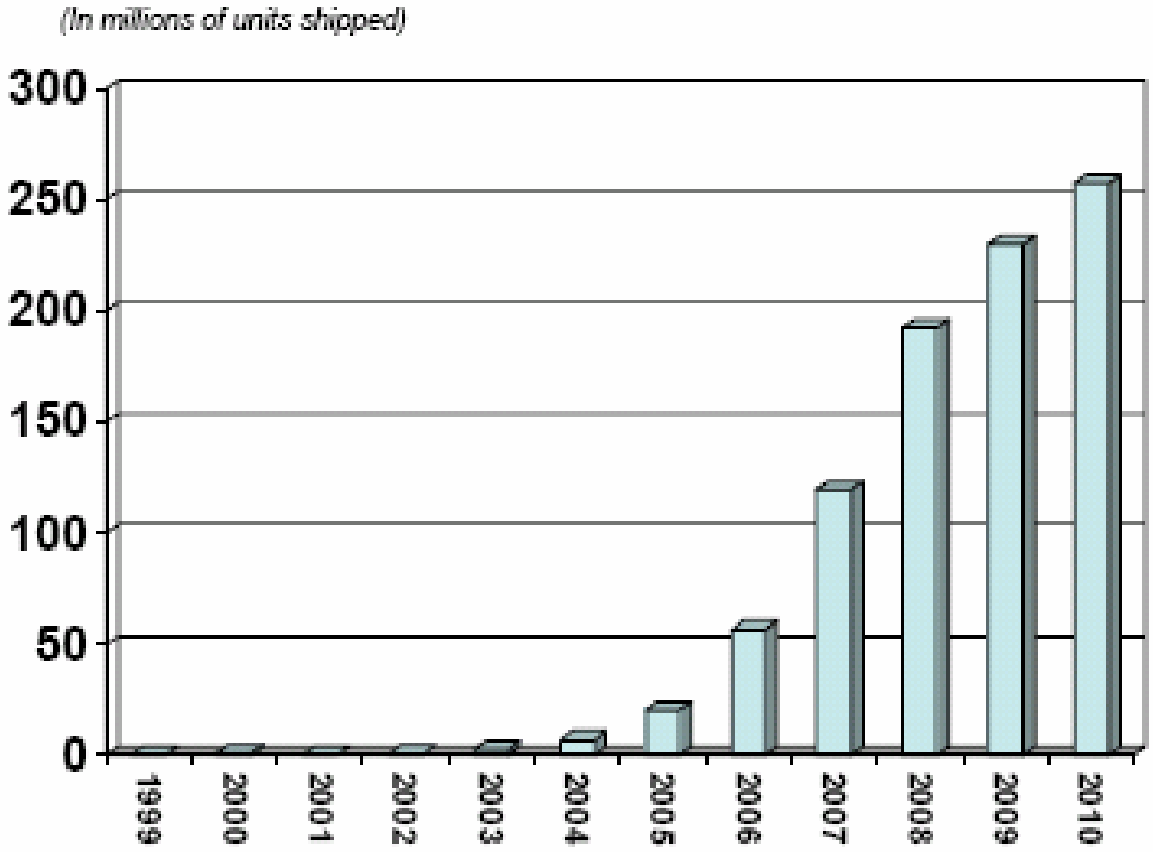
Outline

- Motivation
- TCG architecture
- TPM overview
- TSS overview
- Coding to the TSS
- Mapping to use cases
- Conclusions

Why do we care about trusted computing?

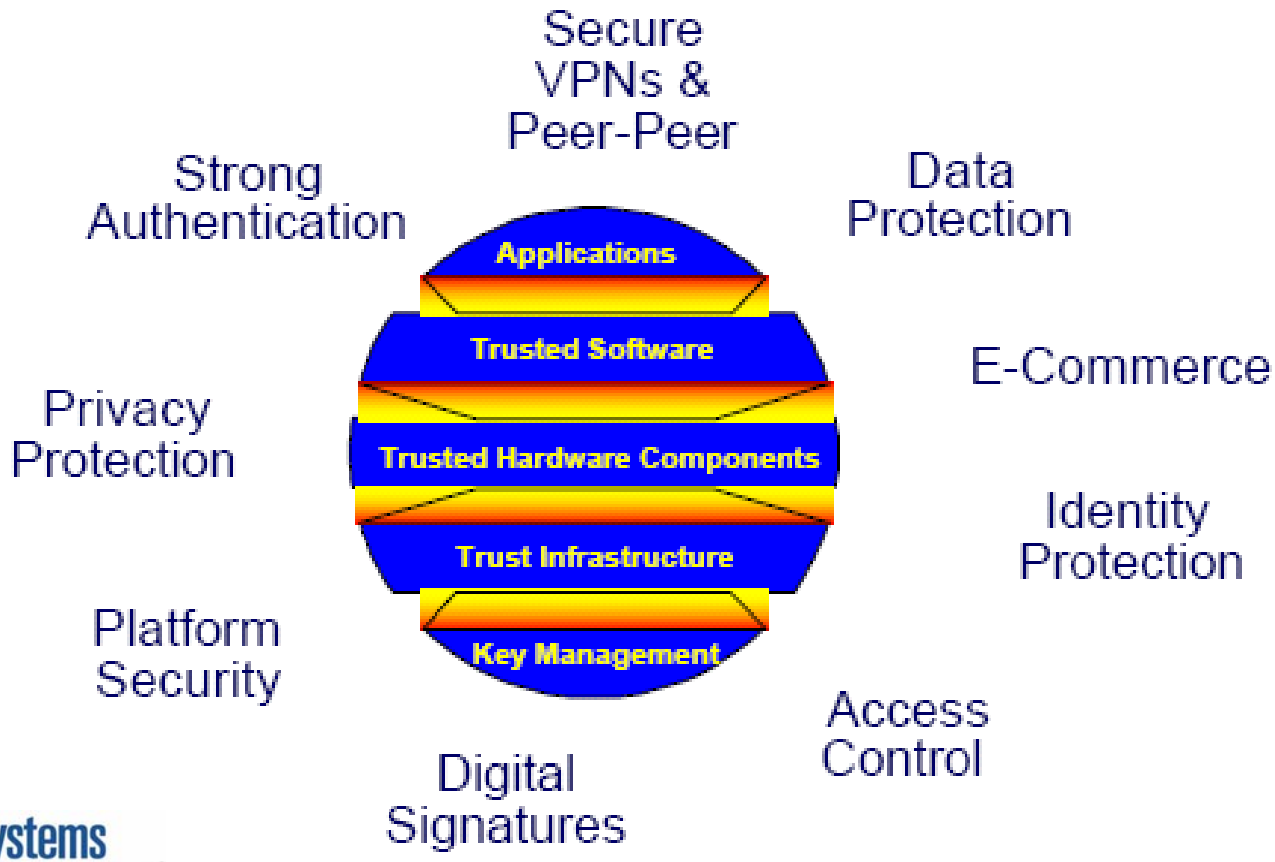
- Trust is key to driving usage of computing devices for security-sensitive applications
- Trusted computing (TC) opens up new usage models
- Wide deployment of TC building blocks

TPM shipment forecast



Source: IDC

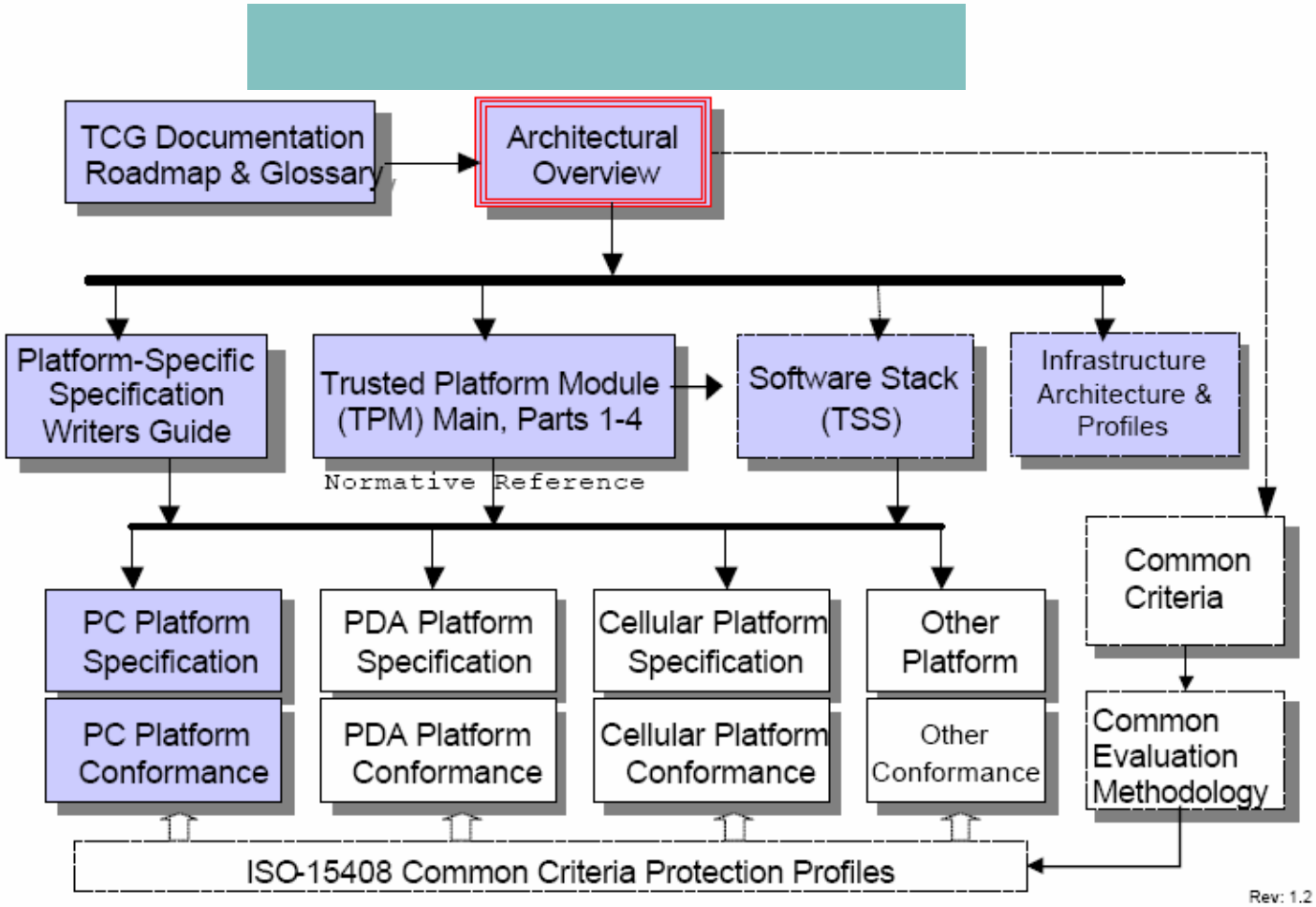
Trusted computing applications



Some motivating use cases

- How do I…
  - Store a key securely, so a user can access it with a password?
  - Back up a key securely, so IT can help the user out when he forgets his password?
  - Ensure that I am communicating with a particular user with access to a particular machine?
  - Make sure my software only runs on a specific machine?
  - Make sure my software runs only on machines in a specific state?
- We'll first do a lot of background, then come back to these use cases

TCG document architecture



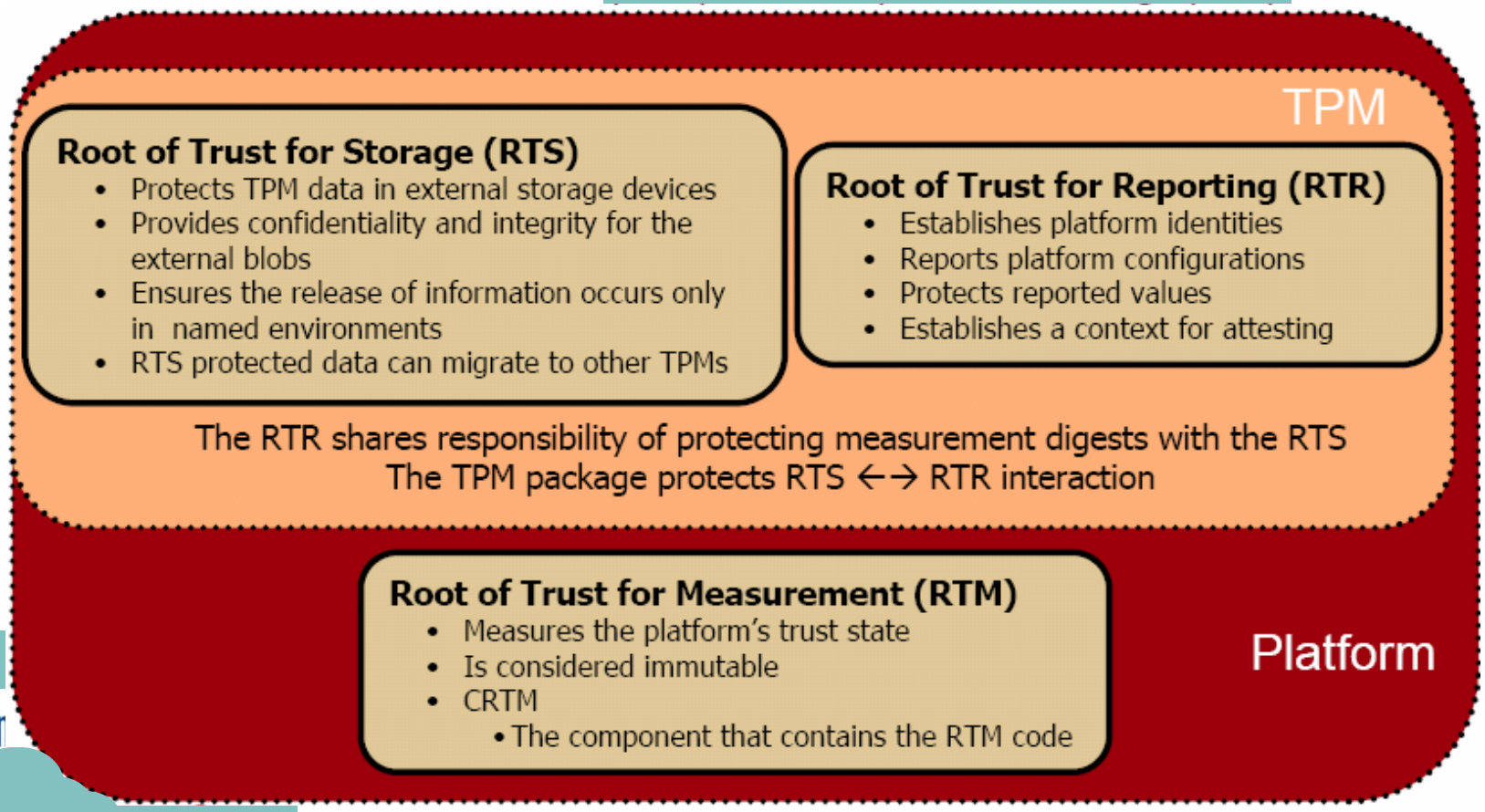
Building on existing building blocks

- There are many components that go into building a “trusted” system
- Great progress has been made building up an ecosystem of trusted computing products and services
- The task ahead of us is to build on that ecosystem

Trusted PC building blocks

- A trusted personal computer (PC) may have many components
  - TCG core roots of trust
    - RTS and RTR inside TPM
    - RTM part of motherboard
  - TCG-enabled BIOS
  - TCG-aware OS
  - TCG Software Stack (TSS)
  - TCG-enabled CSP(s)
  - TCG development tools
  - TCG-enabled applications

Core roots of trust



Trusted Platform Module (TPM)

- Core hardware security component
- Provides
  - Protected storage
  - Protected operations
  - Verifiable reporting of TPM state
  - Controlled access to and use of various keys

Trusted hardware

- To ensure security, the TPM must be securely bound to the platform
- Additional trusted hardware is used for measuring code before running it
- In the PC world, this might be part of the motherboard

TCG-enabled BIOS

- When the PC first boots, the trust in the code is valid as long as the code that is running has been measured
- The BIOS measures the code and performs certain “physical” actions on the TPM

TCG-aware operating system

- If the BIOS measures the operating system (OS) properly, the OS can take advantage of the TPM state
- In embedded systems, this may pose fewer problems than in the PC space

TCG Software Stack (TSS)

- The TSS is a software stack that exposes the functionality of the TPM and provides a common interface to access TPM functionality.
- The main goals of the TSS are:
  - Supply one entry point for applications to the TPM functionality
  - Provide synchronized access to the TPM
  - Hide building command streams with appropriate byte ordering and alignment from the applications
  - Manage TPM resources
  - Release TPM resources when appropriate
  - Manage application use of secrets and keys

TCG-enabled CSP

- Cryptographic Service Providers (CSP) provide standard interfaces to applications for the use of keys (e.g. CAPI, PKCS #11)
- TCG-enabled CSPs abstract away the TPM and TSS
- May or may not reduce TPM functionality

TCG development tools

- Software companies may make development tools available to simplify the writing of TCG applications
- This may make it even easier for application writers to write TCG-enabled applications

TCG-enabled applications

- Applications that ultimately build on the security and trust provided by all of the layers below

TC in embedded systems

- TCG Mobile Phone WG
  - Working on platform-specific specification
- Embedded Linux
  - TrouSerS
  - NTRU CTSS

Embedded trusted computing

- Embedded trusted computing probably won't look exactly like the PC space
- TPM must be there
- TSS probably will be there
- CSP and development tools may or may not be there
- So ... I will focus on the TPM and TSS

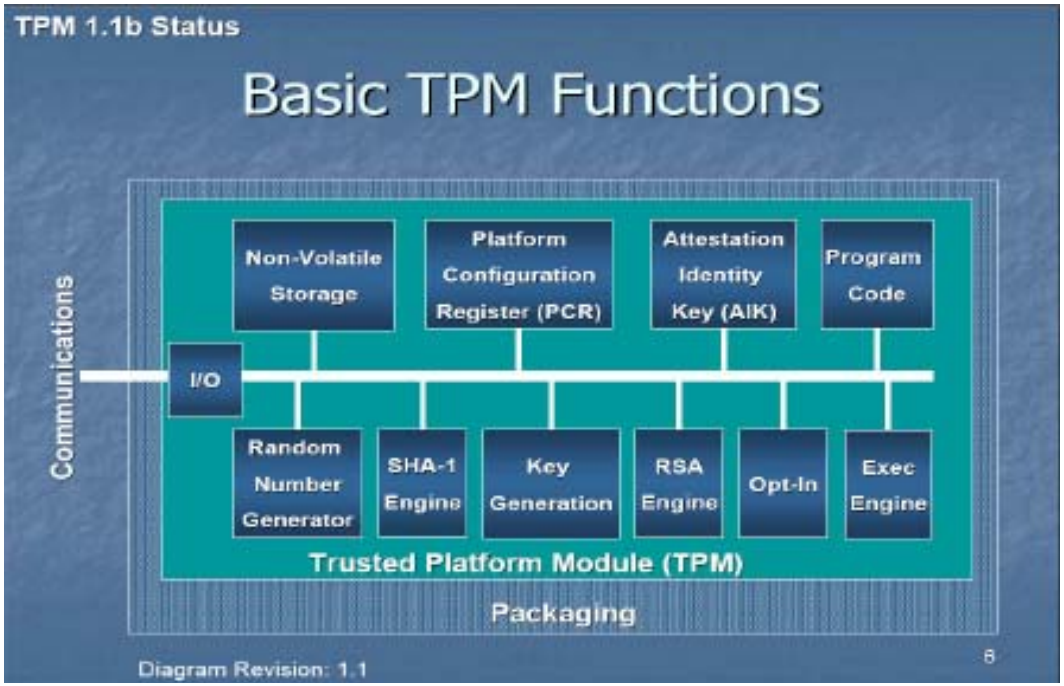
Understanding the TPM

- The TPM specification is rather complex
- Early embedded trusted computing developers would be well served to understand the TPM
- So, here are some of the basics . . .

What makes TPMs special?

- Hardware-based state measuring (root of trust for measurement)
- Hardware-based attestation (root of trust for reporting)
- Fundamental part of the platform
- Wide variety of cryptographic/security functionality
- Robust management interface

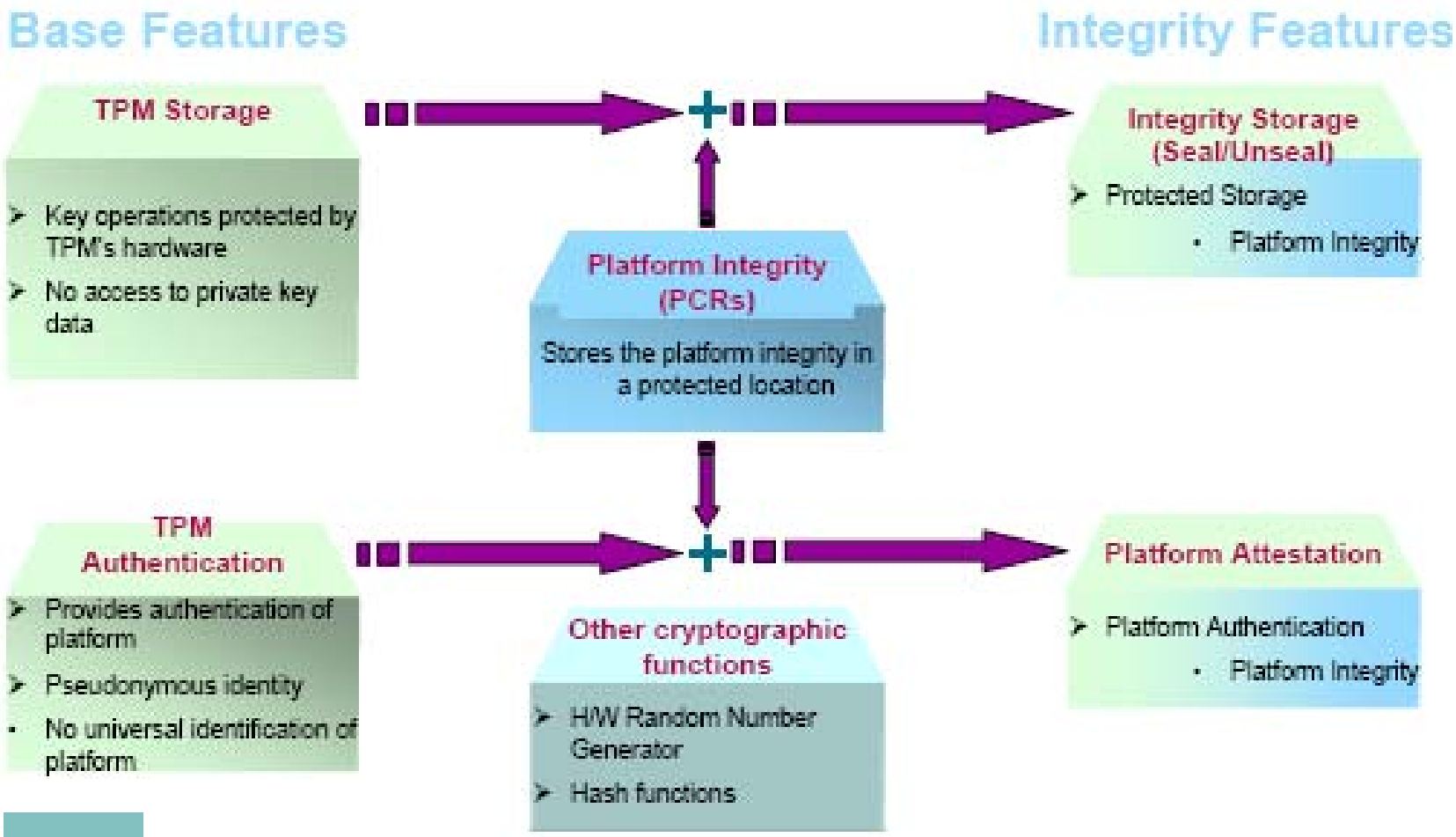
TPM overview



TPM 1.1 core elements

Functional Units	Non-volatile memory	Volatile memory
RNG	Endorsement Key (2048b)	RSA Key Slot-0 . . . RSA Key Slot-9
Hash	Storage Root Key (2048b)	PCR-0 . . . PCR-15
HMAC	Owner Auth Secret (160b)	Key Handles
RSA Key Generation		Auth Session Handles
RSA Encrypt/Decrypt		

TPM features and functions



TPM entities: Owner

- Can control access to the root keys on the TPM (EK and SRK)
  - Protected by use of an authorization secret: essentially a password, stored in TPM tamperproof memory
  - Lockout/response degradation implemented by some manufacturers to make brute force attacks infeasible.
- "Taking ownership" = creating the SRK
  - Requires "physical presence" to be enabled
    - Typically a switch in the BIOS
  - Clearing ownership means that all data protected with the previous key hierarchy cannot be accessed, unless it was previously backed up externally
- In 1.2, owner can perform "delegation"
  - Allows to associate extra passwords with specified operations using specified keys
  - The person who knows the extra password can perform approved actions, but no others.
  - Allows enterprises to have IT as owner, user as privileged operator

TPM entities: Non-owner

- Can create storage (encryption) and endorsement (signing) keys
- Must be under key approved by owner for that use
- Caveat: SRK is generally available for all to use

TPM keys: Endorsement key

- The endorsement key (EK) is the master key that the TPM uses to allow people to take ownership and to prove the security of identity keys.
- EK is a 2048-bit RSA key, often certified during the manufacturing process by a CA
- EK usually unchangeable

TPM keys: Storage keys

- Key hierarchy
  - Each child key is encrypted under its parent.
    - Parents also known as “Storage keys”
    - SRK (Storage Root Key): Top of the tree
  - Keys are migratable or non-migratable
    - Non-migratable includes
      - SRK
      - The parent of any non-migratable key
  - TPM 1.2 has special certifiable migration keys (CMK) that add assurance as to who has access to the private portion of the key

TPM keys: Other keys

- Identity keys: non-migratable signing keys that can be certified by a CA as belonging to a TPM.
- Binding keys for binding,
- Signing keys for signing arbitrary data
- Legacy keys that can both sign and encrypt.
- All keys except the SRK and EK are encrypted by a storage key and stored outside the TPM

TPM objects: PCRs

- Platform Configuration Register
  - In one sense, simply 20 bytes of RAM
  - In another sense, the core of the TPM
  - Used to store hashes ("measurements")
  - Cannot be overwritten, only "extended"
    - Hash new data with current state
  - Can Seal data to set of PCRs, so that it will only be Unsealed if PCRs are in specified state;
  - Can sign and transmit state of a set of PCRs
  - Assume that PCR state is related to software and hardware configuration: then this is very useful

TPM functions: Seal

- TPM_Seal
  - Encrypting data (usually a symmetric key)
  - …using a non-migratable TPM storage key (an RSA key)
  - … so that ONLY that specific TPM can unseal the data.
  - Can be linked to sealing secret (password) and PCR state

TPM functions: Bind

- TSS_Bind, TPM_Unbind
  - Encryption for a binding key that a TPM can use (an RSA key that may or may not be migratable).
  - Not linked to a specific platform
  - Does not use a binding secret and it does not use PCRs.
  - Binding is done outside of the TPM

TPM functions: Migration and quote

- Migration
  - The owner can select keys that the TPM will migrate keys to.
  - Migratable keys can be converted from one "parent" to another.
- Quote
  - A signature using an identity key that attests to the PCR state of the TPM.

TPM 1.2: Capabilities

- CMK - Certifiable migration key
  - TPM can attest they have only been inside the TPM or encrypted for a particular Migration Authority.
  - Enables key backup to other TPMs
- Transport Sessions
  - SSL-like functionality for interaction with the TPM
  - Enables remote administration without eavesdropping

TPM 1.2: Capabilities (2)

- Delegation
  - The ability to give authorization to an entity to do certain things that the owner can do or that a key can do.
  - Enables remote administration by authorized actors
  - Allows IT departments to restrict the damage end-users can do
- DAA – Direct Anonymous Attestation
  - Allow to prove that a command has come from a TPM, without specifying which TPM
    - Uses cryptographic technique known as “group signatures”
  - Partially inspired by European regulatory requirements for privacy

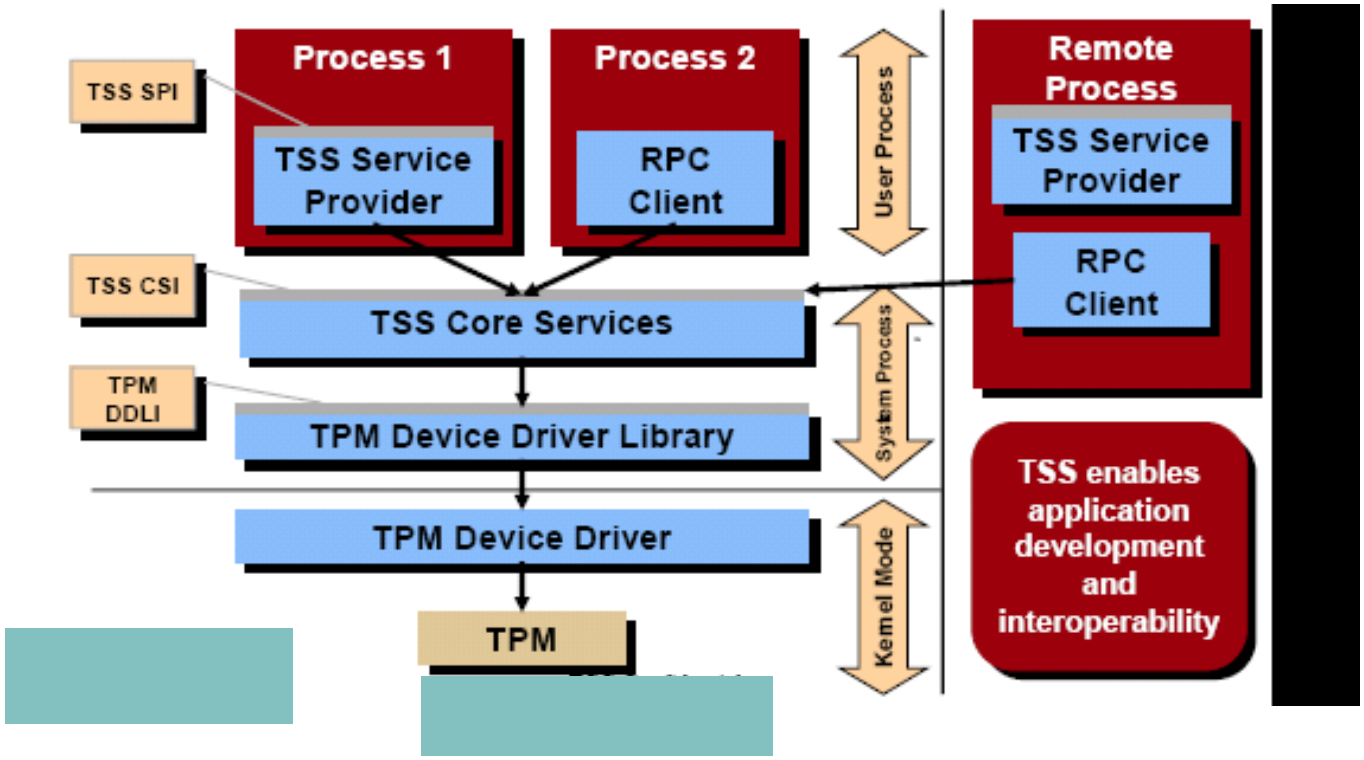
TPM 1.2: Capabilities (3)

- Tick Count
  - A time stamp mechanism.
- Monotonic Counter
  - Non-spoofable, non-resettable counters that can be signed.

Understanding the TSS

- The TSS abstracts some of the TPM complexities away
- If you learn the TPM basics and the TSS API, you can create secure applications

TSS block architecture



TSS Device Driver Library (TDDL)

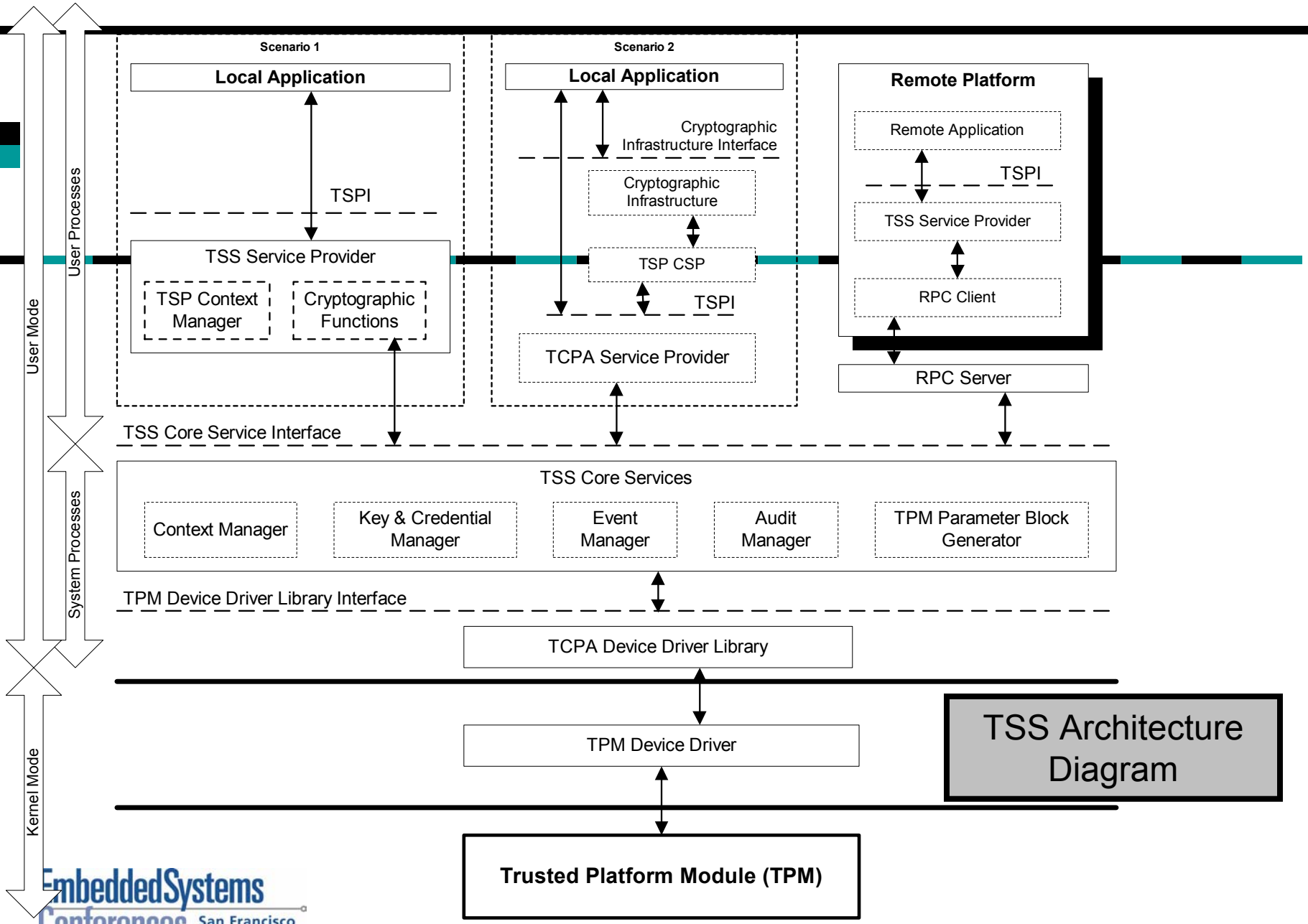
- Creates an abstraction layer hiding OS-specific device driver interfaces from the TCS
- Single point of compatibility for TSS developers
- Allows the TPM vendor to get/set device driver capabilities

TSS Core Services (TCS)

- Parameter Block Generator (PBG)
  - Converts ‘C’ style parameters into TPM format.
- Key and Credential Manager (KCM)
  - Allows the user to alias and persistently store a TPM key.
  - Dynamically swaps keys into and out of the TPM
- Context Manager
  - Allows multiple TSP modules to access TCS simultaneously
  - Performs memory management on a per context basis
- Event Manager
  - Generates, manages and exports “PCR Events”
- Audit Manager
  - Intended to leave records of TPM activity

TSS Service Provider (TSP)

- Exposes TSPI
  - User Friendly API that incorporates object oriented principles
  - Abstracts the underlying protocols and data structures
- TSP Context Manager
  - Allows multiple instances of TSP layer
  - Performs memory management at the TSP Layer
- Public-key cryptography and hashing/HMAC
  - Not all cryptography requires the TPM
  - Performs public-key, hashing and HMAC algorithms to enhance cryptographic security and authorization for the TPM



TSS Architecture Diagram

TSS and key management

- Virtualizes resources used inside the TPM
  - Multiple applications can run simultaneously, each using different keys
  - Applications do not have to manage key load/unload themselves
- To take ownership of the TPM, must write directly to the TSS
  - Currently not possible through higher-layer interfaces such as CSP

TSS for calling TPM functions

- Actions such as Seal are authorized using an authorization secret
  - TSS provides means to enter, cache, and expire the secret
- TPM commands are all formed inside the TSS – they are not exposed directly to the applications

Code samples

- Now, here are some details about how to actually use this stuff . . .

Seal to PCR code

```
int
main(void)
{
    TSS_HCONTEXT    hContext;
    TSS_HTPM        hTPM;
    TSS_HPOLICY     hPolicy;
    TSS_HKEY        hSRK, hSealKey;
    TSS_HENCDATA    hSealData;
    TSS_HPCRS       hPCRs;
    BYTE            wellKnownSecret[] = TSS_WELL_KNOWN_SECRET;
    BYTE            rawData[64];
    UINT32          unsealedDataLength, pcrLength;
    BYTE           *unsealedData, *pcrValue;
    int                                i;

    for (i = 0; i < 64; i++)
      rawData[i] = (BYTE) i;

/* create context and connect to TPM */

    Tspi_Context_Create(&hContext);
    Tspi_Context_Connect(hContext, NULL);
```

Seal to PCR code (2)

```
/* create empty keys and data object */

      Tspi_Context_CreateObject(hContext, TSS_OBJECT_TYPE_RSAKEY, TSS_KEY_TSP_SRK,
                                        &hSRK);
      Tspi_Context_CreateObject(hContext, TSS_OBJECT_TYPE_RSAKEY,
                                        TSS_KEY_TYPE_STORAGE |
                                         TSS_KEY_SIZE_2048 |
                                         TSS_KEY_NO_AUTHORIZATION |
                                         TSS_KEY_NOT_MIGRATABLE,
                                        &hSealKey);

      Tspi_Context_CreateObject(hContext, TSS_OBJECT_TYPE_ENCDATA,
                                        TSS_ENCDATA_SEAL, &hSealData);

/* get TPM object */

      Tspi_Context_GetTpmObject(hContext, &hTPM);

/* set up the default policy - this will apply to all objects */

      Tspi_Context_GetDefaultPolicy(hContext, &hPolicy);
      Tspi_Policy_SetSecret(hPolicy, TSS_SECRET_MODE_SHA1, 20, wellKnownSecret);
```

Seal to PCR code (3)

```
/* create and load the sealing key */

      Tspi_Key_CreateKey(hSealKey, hSRK, 0);
      Tspi_Key_LoadKey(hSealKey, hSRK);

/* seal to PCR values */

    /* set the PCR values to the current values in the TPM */

      Tspi_TPM_PcrRead(hTPM, 5, &pcrLength, &pcrValue);
      Tspi_PcrComposite_SetPcrValue(hPCRs, 5, pcrLength, pcrValue);
      Tspi_TPM_PcrRead(hTPM, 7, &pcrLength, &pcrValue);
      Tspi_PcrComposite_SetPcrValue(hPCRs, 7, pcrLength, pcrValue);

    /* perform the seal operation */

      Tspi_Data_Seal(hSealData, hSealKey, 64, rawData, hPCRs);

    /* unseal the blob */

    unsealedData = NULL;
      Tspi_Data_Unseal(hSealData, hSealKey, &unsealedDataLength, &unsealedData);
```

Seal to PCR code (4)

```
        /* free memory */

        Tspi_Context_FreeMemory(hContext, unsealedData);

/* clean up */

        Tspi_Key_UnloadKey(hSealKey);

        Tspi_Context_CloseObject(hContext, hPCRs);
        Tspi_Context_CloseObject(hContext, hSealKey);
        Tspi_Context_CloseObject(hContext, hSealData);

/* close context */

        Tspi_Context_Close(hContext);

    return 0;
}
```

Sign/verify code

```
int
main(void)
{
    TSS_HCONTEXT    hContext;
    TSS_HHASH       hHash;
    TSS_HKEY        hSigningKey, hSRK;
    TSS_HPOLICY     hPolicy;
    TSS_UUID        srkUUID = TSS_UUID_SRK;
    BYTE            secret[] = TSS_WELL_KNOWN_SECRET;
    UINT32          sigLen;
    BYTE            *sig;
    BYTE            hash[] =
      {0x32, 0xd1, 0x0c, 0x7b, 0x8c, 0xf9, 0x65, 0x70, 0xca, 0x04,
       0xce, 0x37, 0xf2, 0xa1, 0x9d, 0x84, 0x24, 0x0d, 0x3a, 0x89};

/* create context and connect */

       Tspi_Context_Create(&hContext);
       Tspi_Context_Connect(hContext, NULL);
```

Sign/verify code (2)

```
/* create a signing key under the SRK */

     Tspi_Context_CreateObject(hContext, TSS_OBJECT_TYPE_POLICY,
                                    TSS_POLICY_USAGE, &hPolicy);
     Tspi_Policy_SetSecret(hPolicy, TSS_SECRET_MODE_SHA1, 20, secret);
     Tspi_Context_GetKeyByUUID(hContext, TSS_PS_TYPE_SYSTEM, srkUUID, &hSRK);
     Tspi_Policy_AssignToObject(hPolicy, hSRK);
     Tspi_Context_CreateObject(hContext, TSS_OBJECT_TYPE_RSAKEY,
                            TSS_KEY_TYPE_SIGNING |
                             TSS_KEY_SIZE_2048 |
                             TSS_KEY_AUTHORIZATION |
                             TSS_KEY_NOT_MIGRATABLE,
                            &hSigningKey);
     Tspi_Policy_AssignToObject(hPolicy, hSigningKey);
     Tspi_Key_CreateKey(hSigningKey, hSRK, 0);
     Tspi_Key_LoadKey(hSigningKey, hSRK);

    /* open valid hash object */

     Tspi_Context_CreateObject(hContext, TSS_OBJECT_TYPE_HASH,
                                                        TSS_HASH_SHA1,
     &hHash);
```

Sign/verify code (3)

```
/* set hash value and get valid signature */

      Tspi_Hash_SetHashValue(hHash, sizeof(hash), hash);
      Tspi_Hash_Sign(hHash, hSigningKey, &sigLen, &sig);

/* verify signature */

      Tspi_Hash_VerifySignature(hHash, hSigningKey, sigLen, sig);

/* free sig memory, close signing key object and context */

      Tspi_Context_FreeMemory(hContext, sig);
      Tspi_Context_CloseObject(hContext, hSigningKey);

/* close context */

      Tspi_Context_Close(hContext);

    return 0;
}
```

Using this knowledge to write trusted applications

- Given a TPM and TSS, choose correct ways to use functionality to meet security objectives
- Requires some security architecting, but the building blocks are all there
- So, let's look at our original motivating use cases . . .

Secure key storage

- How do I store a key securely, so a user can access it with a password?
  - Select a starting password
    - Set a policy with that password: Tspi_Policy_SetSecret
  - Get the key onto the platform
    - Create a key object with chosen parameters: Tspi_Context_CreateObject
    - Create a new one: Tspi_Key_CreateKey
    - . . . or import a known one: Tspi_Key_WrapKey
  - Optionally let the user change the password
    - Use Tspi_ChangeAuth
  - Store the key blob somewhere
    - Register it in the TSS Key Store: Tspi_Context_RegisterKey
    - Store the key blob somewhere else and load it when needed

Key backup

- How do I back up a key securely, so that IT can help the user out with a forgotten password?
  - Can use remote connection to the TPM . . .
    - Tspi_Context_Connect(hContext, userMachineName);
  - . . . then import key using previous method, keeping a copy of the key
  - Or, use migration capabilities
    - Set up a trusted key to migrate to: Tspi_TPM_AuthorizeMigrationTicket
    - Begin migration process: Tspi_Key_CreateMigrationBlob
    - Complete migration process: Tspi_Key_ConvertMigrationBlob
  - That's it!

User and machine authentication

- How do I ensure that I am communicating with a particular user with access to a particular machine?
  - Choose TPM authentication method
    - Decryption of a challenge: Tspi_Data_Unbind
    - Signature of a challenge with selected configuration: Tspi_TPM_Quote
  - Set up a key for the user on a particular TPM
    - Use previous methods
    - Perhaps ensure that the key is non-migratable: TSS_KEY_NOT_MIGRATABLE
  - Require that the user use the key as a means to authenticate
    - Could leverage existing authentication mechanisms
      - Smart cards
      - VPN
      - Digital signature
    - Could use a new protocol if desired

Binding to a specific platform

- How do I make sure my software only runs on a specific machine?
  - Use the previous authentication mechanisms, but embed them in the application
    - Require a signature from a key tied to the platform before continuing: Tspi_Hash_Sign, Tspi_Hash_VerifySignature
  - Encrypt key data to the platform
    - Seal a symmetric key that the application needs to function: Tspi_Data_Seal, Tspi_Data_Unseal

Binding to a platform state

- How do I make sure my software only runs on machines in a specific state?
  - Leverage secure use of PCRs
    - Could involve secure boot
    - Could involve use of locality
  - Leverage platform credentials
    - Assume that each EK has a certificate
    - Create identity keys for your application: Tspi_Key_CreateKey, Tspi_TPM_CollateIdentityRequest, Tspi_TPM_ActivateIdentity
    - Certify other keys (e.g. an encryption key) if necessary: Tspi_Key_CertifyKey
  - Require platform authentication of state before allowing software to be run
    - Encrypt software and require certified key to decrypt when in correct state using previous methods
    - Require signature of state before continuing application: Tspi_TPM_Quote

Conclusions

- TCG technologies provide a very rich set of functionality to implement security features
- For embedded applications, perhaps the TSS interface is the appropriate one to use
- Early application writers will need to understand a few of the specifics of TCG technologies
- The increasing deployment trend of TCG technologies will allow for more and more trusted applications to come into existence

Questions?

Contact Info:

Ari Singer, NTRU Cryptosystems

asinger@ntru.com